A Simple Method for Extracting Models from Protocol Code

David Lie, Andy Chou, Dawson Engler and David Dill
Computer Systems Laboratory
Stanford University
The Validation of Modern Systems

• Our ability to build complex systems has outstripped our ability to verify and validate them

• There are several reasons for this:
  – It is difficult to get good coverage of large systems
  – Traditional testing is tedious and very costly
  – Applying formal methods is hard

• Our goal: Create tools to facilitate the application of formal methods
Model Checking

• It is a formal method, but with more pragmatic goal of finding bugs rather than proving system correctness

• System is modeled as a state machine
  – States are explored
  – Correctness is checked at every state
  – If the state space is too big, an estimate on the number of missed states is performed
  – This places some bound on the possibility of missing a bug
Problems with Model Checking

- Traversing the state space requires a lot of memory to remember visited states

- Constructing the models is difficult and error-prone
  - Details should be abstracted
  - Model needs to be scaled down

- Manually constructing models causes:
  - Errors to be introduced by human modeling
  - Models to miss implementation changes (Drift)
Case Study: FLASH

- Distributed shared memory multiprocessor
- Cache-coherence protocol runs in node controller firmware
- Many race conditions and corner cases due to concurrency and distributed state

Interconnection Network

Mem 0 | NC 0 | CPU 0
Mem 1 | NC 1 | CPU 1
Mem 2 | NC 2 | CPU 2
Simplifying the Application of Model Checking

- Use compiler support to extract a model of the software implementation
The Tools

- Leverage two existing tools:
  - xg++ extensible compiler
  - Mur$\varphi$ model checker
xg++ Compiler

• Modified g++ that allows users to add extensions that operate on the Abstract Syntax Tree (AST)
  – Analyze or modify AST
  – Several extensions can be run one after another as separate passes

• Users write extensions in a language called *Metal*
  – Pattern matching language
  – Arbitrary actions taken when patterns match
The Mur$\phi$ Model Checker

• Models are described by a set of functions called “rules”
  – Preconditions
  – Assertions
  – Transitions

• Model checker
  – Starts in an initial state
  – Looks for rules with applicable preconditions
  – Checks that assertions are satisfied
  – Executes transitions to compute the next state
Example \textit{Mur}\textsubscript{ϕ} Code

Rule "PI Local Get"
\begin{itemize}
\item Cache.State = Invalid \& !Cache.Wait
\end{itemize}

Begin
\begin{itemize}
\item Assert !DH.Local "PILocalGet:2"
\item Assert DH.Head \& !DH.List \& DH.Real=0 "PILocalGet:1"
\end{itemize}

DH.Pending := true;
Cache.Wait := true;

Send Request(Home, DH.HPtr, Get, Home, void);
End;

- Rule has a precondition which indicates when it can be applied
- Assertions check for correctness
- Variable assignments compute new values
- State also includes network; this is also a state transition
The Extraction Process

- Two configuration components specified by user
  - State variables
  - Translation Patterns

![Diagram showing the extraction process]

- Protocol Code (Implementation)
- State Variables
- Translation Patterns
- xg++ compiler
- Correctness Properties
- Protocol Model
- Hardware Model
- Initial State
- Murφ
- Error List
A Key Observation: Handlers map to Rules

- A handler in FLASH is triggered by an event
  - A request from the processor
  - A request from the network

- Rules in Mur$\phi$ execute when precondition is satisfied
  - Processor needs a piece of memory
  - A message arrives from another node controller

- A handler in FLASH is equivalent to a rule in Mur$\phi$
Specifying State Variables *(Metal Slicer)*

- This allows users to indicate the important state variables and functions to the compiler

- Compiler uses “Program Slicing” algorithm
  - Determines which statements should be extracted to create the Protocol Model

- Example, slice out the network header field:
  ```
  pat length = { packet.length };
  pat sends = { NI_SEND(data) };
  ```
Specifying Translation Patterns (Metal Printer)

• The user tells the compiler about places where it can
  – Perform abstractions to reduce state space
  – Insert extra checks to tighten correctness properties

• Example, convert two different values into one:

  \[
  \{ \text{len}_\text{cacheline} \} \mid \{ \text{len}_\text{word} \} \implies \{ \text{emit("len}_\text{data")}; \}
  \]
Other *Metal Printer* Examples

- **Insert code to check network sends:**
  ```
  { NI_SEND(argument); } ==> 
  if ( is_constant(data) != 0) 
      emit("assert(packet.length = len_data);"); 
  else 
      emit("assert(packet.length = len_nodata);"); 
  emit("ni_send(%t);", argument); 
  ```

- **Abstract away bit operations:**
  ```
  { packet ^= ( new_opcode ^ old_opcode ); } 
  ==> { 
      emit("packet.opcode = Opcode_substitute(%t, %t, 
              packet);
          ni_send(%t);", old_opcode, new_opcode); 
  } 
  ```
Protocol Code

void PILocalGet(void) {
    // Boiler Plate Setup Code
    Load_From_memory (&directory)
    // Debug code
    printf ("Entering PILocalGet\n");
    packet.length = len_cacheline;
    if (!directory.Pending) {
        if (!directory.Dirty) {
            directory.IO = 1;
        
            NI_SEND(data);
            directory.Pending = 1;
        } else {
            ASSERT(!directory.List);
            Gather Statistics();
        }
    } else {
        ASSERT(!directory.List);
        Gather Statistics();
    }
}

Protocol Model

Rule "PI Local Get"
Cache.State = Invalid & !
Cache.Wait ==> Begin

packet.length := len_data;
if (!directory.Pending) then
    if (!directory.Dirty) then
        assert(packet.length = len_data)
        ni_send (packet);
    directory.Pending := 1;
else
    assert(!directory.List);
Combining the Other Components

- Extracted Protocol Model is combined with Correctness Properties, a Hardware Model and an Initial State
Correctness Properties

• Correctness Properties are defined by
  – Assertions in the rules (extracted)
  – Global Invariants (manually added)

• Global Invariants:
  – They are checked at every state
  – Example:
    • Don’t overflow network queues
    • Don’t invalidate a processor who has data in the exclusive state
The Hardware Model

R10K Processor Model

Processor Interface (PI)

Protocol Model

I/O Interface (IO)

I/O Subsystem Model

Network Interface (NI)

Network Model

Murφ Model
Verification Results

- \textit{Mur}_\phi \textit{ outputs an error trace that aids in finding errors}
- The automatic extraction reduces the manual effort as well as making the model more faithful to the implementation
The Error List

- If a bug is found
  - The error is stated
  - A trace of all visited states and executed rules is given

- Whether a bug is found or not
  - $\text{Mur}_\phi$ gives number of states explored
  - For us, our largest models had about 6 million states
Example of Gory Bug

Node 2 is on the sharing list even though it does not have the line.
# Reduction in Manual Labor and Increase in Effectiveness

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Pointer</td>
<td>6</td>
<td>12K</td>
<td>1100</td>
<td>1000</td>
<td>99</td>
</tr>
<tr>
<td>Bitvector</td>
<td>2</td>
<td>8K</td>
<td>700</td>
<td>1000</td>
<td>100</td>
</tr>
<tr>
<td>RAC</td>
<td>0</td>
<td>10K</td>
<td>1500</td>
<td>1200</td>
<td>119</td>
</tr>
<tr>
<td>Coma</td>
<td>0</td>
<td>15K</td>
<td>2800</td>
<td>1400</td>
<td>159</td>
</tr>
</tbody>
</table>
Conclusion

• A method to make the application of model checking easier and more effective was developed

• The extracted protocol model is both easier to produce and more faithful to the implementation

• Method relies on a user who understands the details of the system being modeled